Possibilities of Semi-automated Generation of Scenarios for Simulation Testing of Software Components

Tomas Potuzak*[1], Richard Lipka[2]

Department of Computer Science and Engineering, Faculty of Applied Sciences, University of West Bohemia
Univerzitni 8, 306 14 Plzen, Czech Republic

*[1]tpotuzak@kiv.zcu.cz; [2]lipka@kiv.zcu.cz

Abstract

Component-based software development where the applications are constructed from reusable software components (possibly from different manufacturers) becomes an important trend in software engineering. Since the testing of the components is important, we developed the SimCo tool, which enables simulation testing of the components directly (i.e. without necessity to create their simulation models). In this paper, we discuss the possibilities of the semi-automated generation of the testing scenarios describing the course of the testing of the components. Two approaches are considered – the use-case-based scenarios generation and the interface-based scenarios generation. The former approach utilizes the use case descriptions of the tested components written in natural language for the generation of the testing scenarios. The latter approach utilizes known (public) interfaces of the tested components for the generation of the skeletons of the testing scenarios.

Keywords

Testing; Testing Scenarios; Semi-automated Generation; Use Case Description; Natural Language; Component Interface

Introduction

The component-based software development is a spreading trend since the turn of the century in both academic sphere and industry. Using this approach, an application is constructed from individual and isolated parts called *software components*. Each software component provides services through a well-defined interface. These services can be used by other software components of the application. On the other hand, the software component can (but does not have to) require services provided by other components of the application. A single component can be used in multiple applications, which leads to a better reuse of the existing code. Moreover, a single application can be constructed from components developed by different manufacturers (Szyperski, Gruntz, and Murer 2000).

Similarly to all other types of software development, the testing is very important using the software components. Besides the testing of the functionality of the particular software components, it is also necessary to test their extra-functional properties (e.g. the size of the memory consumed by the calculations of a software component) and quality of services (e.g. time required for execution of services of a software component). Some other examples can be found in (Becker, Koziolek, and Reussner 2009) and (Heam, Kouchnarenko, and Voinot 2010).

During our previous research, we have developed a simulation tool called SimCo, which enables to perform the simulation testing of particular software components, sets of software components, or entire component-based applications (Potuzak, Lipka, Brada, and Herout 2012). The software components can be tested in the SimCo simulation tool directly – it is not necessary to create their (potentially incorrect) models (Potuzak, Lipka, Brada, and Herout 2012). The course of the testing is described in simulation scenarios, which are loaded by the SimCo simulation tool. Currently, these scenarios are created manually, which is a lengthy and error-prone process. Moreover, some important features of the software components may be not tested at all just because of an omission by the creator of the scenario.

In this paper, we discuss possible approaches to semi-automated generation of the scenarios for the simulation testing. The options depend on the available information about the tested software components. If we are developers of the components, their source codes, UML models, and all other resources are available. However, if software components are developed by another manufacturer, only the interface of the software component and its description can be known. In the former case, it is possible to utilize the use-case diagrams and

descriptions or other behavioral descriptions of the software components for semi-automated generation of the testing scenarios. In the latter case, the known interfaces of the components can be utilized.

Software Component Description

In order to make further discussion more clear, we will briefly describe the component-based approach.

Basic Notions

A software component is a black box entity with well defined interface with no observable inner state. The software components are parts of the third party composition, since the author of a component-based application can be different from the author(s) of the particular software components (Szyperski, Gruntz, and Murer 2000).

A component model specifies the features, behavior, and interactions of the particular software components. A component framework is then a specific implementation of a component model (Potuzak, Lipka, Brada, and Herout 2012). Naturally, there can be more than one implementation. So, there can be multiple component frameworks for a single component model. Also, there exist several widely used component models (OSGi 2009; Bures, Hnetynka, and Plasil 2006).

The various component models can have very different features and approaches to solving of various issues of the software components. Hence, the transformation of an existing software component from one component model to another is often quite difficult. Therefore, the SimCo simulation tool is based on one component model only – the OSGi (OSGi 2009).

The OSGi Description

The OSGi is a dynamic component model for Java. There are several implementations of the OSGi model (i.e. OSGi frameworks) (OSGi 2009). The dynamic nature of the OSGi model means that it is possible to install, start, stop, and uninstall software components without the necessity to restart the OSGi framework (Rubio 2009). Various OSGi frameworks are widespread in different industry areas such as vehicles, cell phones, PDAs, and so on as well as in academic sphere. That is the reason why the OSGi was selected for the SimCo simulation tool (Potuzak, Lipka, Brada, and Herout 2012).

In the OSGi model, the software components are referred as *bundles*. Since the OSGi model is designed for Java, an OSGi bundle has the form of a standard *.jar* file with a special manifest file. This file contains information about the features of the OSGi bundle, about its provided and required services, and so on. The *.jar* file can contain any number of classes. Hence, a bundle can provide arbitrary complex functionality (Potuzak, Lipka, Brada, and Herout 2012).

As the interface of the bundle (which is externally visible unlike its inner state and behavior), standard Java interfaces are used. Each OSGi bundle registers the interface in the OSGi framework and provides its implementations. So, the services provided by the bundles have the form of methods (OSGi 2009). The registration of the services can be performed directly in the source code of the bundle or by the Component Service Runtime for declaratively specified services (Potuzak, Lipka, Brada, and Herout 2012).

Simulation and Testing

Generally, there are two types of discrete simulations considering their simulation time – the *time-stepped* simulations and the *discrete-event* simulations. In a time-stepped simulation, the simulation time is subdivided into sequence of equally-sized time steps. In each time step, the entire simulation state is recomputed. In a discrete-event simulation, the simulation time is subdivided into sequence of time-stamped events representing incremental changes of the simulation state (Fujimoto 2000).

In the SimCo simulation tool, the discrete-event simulation is utilized. Hence, it is described more thoroughly in the following subsection.

Discrete-event Simulation Description

In a discrete-event simulation, the simulation time between two succeeding events can be arbitrary short or long. However, the simulation does not wait, since there are no actions between the events to be performed. The actions are associated with events as well as the time stamps. Hence, the simulation time "jumps" from a time stamp to another (Fujimoto 2000).

The events are handled by a so-called *calendar* with list of events to be performed ordered by their time stamps. When the simulation is started, the calendar removes the first event from the list, sets the simulation time according to its time stamp, and performs its action (i.e. an incremental change of the simulation state). The action can cause the creation of a new event, which is put to the list of events according to its time stamp. When the event is processed, the currently first event is removed from the list of events

and the entire process repeats (Fujimoto 2000).

The simulation ends when a certain simulation time is achieved or there are no events left in the list of events (Potuzak, Lipka, Brada, and Herout 2012).

Simulation Testing Description

Generally, a simulation can be used for testing of various systems. However, we will focus on the simulation testing of the software components.

For the simulation testing, two approaches can be used. First possibility is to develop a model of the software component, which shall be tested (Becker, Koziolek, and Reussner 2009). The advantage of this approach is that the model is suited for the simulation. The disadvantage is the necessary and potentially error-prone creation of the model. Since the model usually does not incorporate all aspects of the original software component, it is also possible to unintentionally omit features, which can be important for the testing (Potuzak, Lipka, Brada, and Herout 2012).

Second possibility is to use the software component in the simulation instead of its model. The advantage of this approach is that the component is tested directly and there are no errors introduced by the creation of the model of the component. However, the software component is not suited for the running in a simulation. Hence, the simulation must be adapted to allow running of the software components in a way that the software component is not aware it is situated in a simulated environment (Potuzak, Lipka, Brada, and Herout 2012).

The simulation testing and the testing in general can be also divided according to the knowledge of the tested software component. If its source code is known, it can be used for preparation of the course of the testing. That is so-called *white box* testing. If the source code and the inner behavior of the tested component are unknown, the course of testing must be prepared using some available specifications of the component. That is so-called *black box* testing (Sapna and Mohanty 2008).

Testing Scenarios Description

In either case, the simulation testing lies in subjecting of the tested software component or its model to a set of stimuli and observation of the corresponding reaction. The stimuli must be kept in reasonable constrains depending on the tested software component. However, for complex software components, it can be difficult to unfeasible to cover all its stimuli and reactions. Hence, an efficient set of stimuli, which represents well the theoretical total cover of all possible stimuli, should be created and then used during the testing (Sapna and Mohanty 2008).

The course of the testing is described in so-called *scenarios*. Considering the simulation testing and the discrete-event simulation, the scenarios contain the events, which shall be performed during the testing. Further information including the configurations of various aspects of the simulation and the testing can be included as well. The event corresponds to subjecting of the tested software component or its model to a stimulus (see above).

More specifically, in case of the simulation testing of software components, the events correspond to the invocations of services with various parameters and observation of the reactions of the software component (i.e. the return value, invocation of a service of a different software component, etc.). The observation can be implemented within the same event, within another event, or using another mechanism separated from the event processing such as logging or saving of the simulation testing states including the values of all observed parameters.

SimCo Simulation Tool

The SimCo simulation tool, which we developed, enables simulation testing of individual software components, sets of components, or entire component-based application. The tested software components are directly incorporated in the simulation. Hence, it is not necessary to create their models. On the other hand, the simulation environment must be constructed in a way that the tested software components are not aware that they are running in a simulation environment. The SimCo simulation tool itself is constructed from software components as well in order to enable its simple modifications and extensions (Lipka, Potuzak, Brada, and Herout 2013). For this reason, there are several types of software components in the SimCo simulation tool (see below).

Component Model Used by SimCo Simulation Tool

The SimCo simulation tool is based on the OSGi component model. It is running in an OSGi framework. Hence, the set of software components, which can be tested using the SimCo simulation tool, is limited to the OSGi bundles.

However, this restriction is not too hard because of the widespread of the OSGi in both academic sphere and

the industry (Lipka, Potuzak, Brada, and Herout 2013).

Software Component Types in SimCo Simulation Tool

In the SimCo simulation tool, there are four types of software components – the *core*, the *real tested*, the *simulated*, and the *intermediate* components (Lipka, Potuzak, Brada, and Herout 2013).

The core components represent the SimCo simulation tool itself. These components ensure the running of the simulation and provide all necessary supplementary functions such as scenarios loading, logging, measuring and storing of observed parameters, visualization of the simulation, and so on. The most important core component is the *Calendar*, which interprets the events from the list of events and ensures the advancement of the simulation in time (Lipka, Potuzak, Brada, and Herout 2013).

The real tested components are the software components, which are tested using the SimCo simulation tool. The source code of the components may be known or unknown, which implies white or black box testing. Although the goal is not to manipulate the tested components in any way, there are issues with the running of the real components in the simulated environment (e.g. discrepancy of the simulation and real time, undesirable network communication, etc.). These issues can be solved using various means including aspect oriented programming and even changing of the tested components' bytecode. However, a further discussion of this matter is outside the scope of this paper. For more details, see (Potuzak, Lipka, Brada, and Herout 2012) and (Lipka, Potuzak, Brada, and Herout 2013).

The simulated components together with the core components form the simulation environment for the real tested components. These components provide services, which are required by the real tested components. The simulated components can be also used for speedup of the simulation, since they do not have to perform all calculations, which the software components mimicked by the simulated components would do. For this purpose, sets o pre-recorded return values or random numbers generators can be used depending on the situation. The last, but very important role of the simulated components is to ensure the control of the simulation by the SimCo simulation tool. For this purpose, all invocations of the services of the simulated components are performed using the events and the calendar (Lipka, Potuzak, Brada, and Herout 2013).

The intermediate components are introduced in order to maintain the control of the simulation by the SimCo simulation tool even between two real tested components. It is desirable to perform all invocation of the services using the events and the calendar. When a real tested component invokes a service of a simulated component, the control is accomplished using the simulated component (see above). However, if a real tested component invokes a service of another real tested component, this invocation would be missed by the SimCo simulation tool. Hence, an intermediate component is used as the proxy for the real tested component. The intermediate component provides the same services as the real tested component and all invocations of the services of the real tested component are in fact handled using this proxy. The intermediate component ensures the using of events and the calendar and performing the invocation of the required services on the real tested component (Lipka, Potuzak, Brada, and Herout 2013).

An example of three types of the software components of the SimCo simulation tool (except the core components) is depicted in Fig. 1. There are two real, one intermediate, and three simulated components.

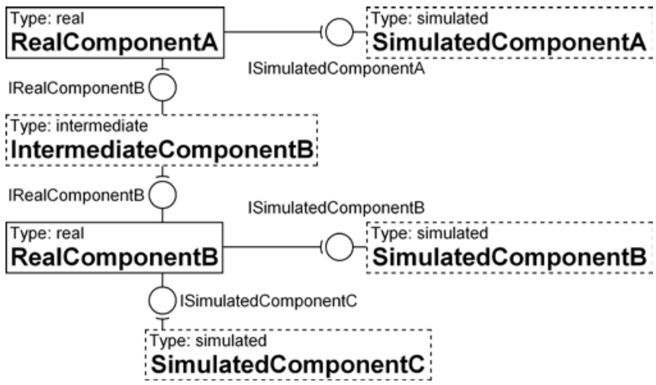

FIG. 1 AN EXAMPLE OF THREE COMPONENT TYPES

Scenarios for SimCo Simulation Tool

The scenarios for the SimCo simulation tool are XML files with defined structure containing the configuration of the simulation and testing and the events, which are introduced into the simulation (Lipka, Potuzak, Brada, and Herout 2013).

The events in the scenario can be described separately, which means that each event is describes as one record incorporating the time stamp and an action. It is also possible and often practical to use a generator of repeating events using random numbers generators and a convenient stochastic distribution (Lipka, Potuzak, Brada, and Herout 2013).

The scenarios could also contain information about the

expected reaction of the tested software components to the introduced events (e.g. expected values of the observed parameters after an invocation of a service on a software component). Currently, this approach is not used. The observation and evaluation of the values of the important parameters are handled separately using logging and probes prepared for the testing of specific software components (Lipka, Potuzak, Brada, and Herout 2013). However, the incorporation into the scenarios is now under consideration.

Currently, the scenarios are created manually, which is a lengthy and error-prone process. Hence, we are searching for a viable approach to semi-automated or even automated generation of the scenarios for the SimCo simulation tool. Several existing approaches are discussed in following section.

Related Work

The idea of the semi-automated or even fully automated generation of testing scenarios based on various sources is not new, although not focused on the testing of the software components. The research in this area was conducted at least since 1970s (Bauer and Finger 1979) and there are works describing the research conducted recently (Shirole and Kumar 2013). The existing approaches have in common that they are not based on the source code of the tested application, but rather on the description of its behavior or the requirements on the application. Some examples are briefly discussed in following subsections.

Utilization of UML Behavioral Diagrams

Many approaches are based on UML behavioral diagrams such as activity diagram, sequence diagram, and state machine diagram (Shirole and Kumar 2013). An approach based on activity diagrams can be found for example in (Sapna and Mohanty 2008), where these diagrams are used to represent concurrent activities. The exploration of the activity diagrams is then used for generation of the testing scenarios. However, for large applications and multiple use cases, the exploration of all possible flows in the activity diagrams would produce too many scenarios. Hence, the constraints derived from the application domain are used to discard scenarios, which are illegal or irrelevant (Sapna and Mohanty 2008).

Another approach is based on case activity diagrams, which are constructed from multiple UML use case diagrams in order to consider the concurrency of particular use cases. The generation of the testing scenarios is then based on the exploring of the case activity diagrams (Hou, Wang, Zheng, and Tang 2010).

In (Yongfeng, Bin, Minyan, and Zhen 2009), multiple types of UML diagrams are used in the testing scenarios generation process. However, the execution paths, which are directly used for the scenarios generation, are extracted during exploration of a layered activity diagram describing the workflow of the tested application (Yongfeng, Bin, Minyan, and Zhen 2009).

Utilization of User Interface

Since the user interface usually enables access to the majority (if not all) functions of the tested application, it can be also used for generation of testing scenarios. In (Liu and Shen 2012), the interface scenarios are created by inducing inputs to the user interface and pairing them with the resulting outputs. The interface scenario is then represented using finite state machine and compared to the formal specification of the tested application (Liu and Shen 2012).

An opposite approach is described in (Naseer, Rehman, and Hussain 2010). It is focused on the testing of software components created using .NET platform. In this case, the absence of a user interface of a component providing some services is stressed as a problem for its testing. Hence, the reverse engineering is used for obtaining of the classes, methods, and attributes of the component and a basic graphical user interface for the component is generated. This interface can be then used for testing of particular classes and methods of the component (Naseer, Rehman, and Hussain 2010).

Utilization of Natural Language Specification

An appealing approach is to use the specification of the tested application written in natural language for automated generation of the testing scenarios. However, due to the ambiguity, poor understandability, incompleteness, and inconsistency of the natural language (De Santiago and Vijaykumar 2012), this approach is still very difficult to implement. Often, some restrictions are used to overcome the difficulties.

An example can be found in (Somé and Cheng 2008) where only a restricted form of natural language is used for use case descriptions. From these descriptions, control-flow-based state machines are generated for particular use cases and interconnected into a global system level state machine. By exploration of this state machine, the testing scenarios can be generated (Somé and Cheng 2008).

Semi-automated Scenarios Generation

The majority of the mentioned approaches has in common that the testing is performed by the manufacturer of the application. So, all the descriptions of the application, UML models, and even source code can be available during the preparation of the testing scenarios. Considering the testing of software components, this may not be the case in all instances. Often, it is necessary to test not only software components, which we developed, but software components produced by different manufacturers as well. In that case, the UML diagrams, behavioral descriptions, and so on may or may not be available.

Therefore, for the generation of the testing scenarios for the SimCo simulation tool, we consider both possibilities. Naturally, the less information is available about the tested software component, the smaller part of the testing scenarios can be created automatically. Two considered approaches are described in following subsections.

Use-Case-based Scenarios Generation

The first approach, which we are considering, is the generation of the testing scenarios from the UML use case descriptions written in natural language, assuming this information is at our disposal. This approach is similar to (Somé and Cheng 2008), but utilizes different restrictions and different approach to the generation of the testing scenarios.

For the analysis of the use case descriptions, we utilize the Procasor tool (Simko, Hauzar, Bures, Hnetynka, and Plasil 2013). The Procasor tool is able to transform the use case descriptions enriched by annotations into an overall behavioral automaton (OBA) (Simko, Hnetynka, Bures, and Plasil 2012a). The annotations are used to describe the flow of the program (e.g. branching, jumps, etc.) and its temporal dependencies (i.e. order of program's actions). So, direct manual transformation of the descriptions written in natural language into more formal models such as activity diagrams (Shirole and Kumar 2013) is not necessary. Nevertheless, the descriptions must be written using the rules described in (Cockburn 2000) and the text must be enriched by the annotations.

The overall behavioral automaton, which is result of the transformation of the use case descriptions with the Procasor tool, can be used for automatic verification of the consistency of the use case descriptions (Simko, Hnetynka, Bures, and Plasil 2012a; Simko, Hnetynka, Bures, and Plasil 2012b). The annotations enriching the descriptions are used by construction of the OBA. For the generation of the testing scenarios, we intend to use new annotations interconnecting the created OBA with the implementation (i.e. source code) of the software components.

There are several new annotations utilized for the generation of the testing scenarios. The *method* annotation is used for interconnection of an action of the user or the system described in a use case description and a corresponding method invocation where it is possible. If they are known, the parameters, the return value, and the invoker should be written in the annotation. The *create-use* annotations is used for expression that in a step of the use case description, an artefact (e.g. a variable, an object, etc.) is created or used. The *create-use* annotations are not entirely new. They are used by the Procasor for checking of the correctness of the use case description. However, we can utilize them for the generation of the testing scenarios as well. The *efp* annotation can be used for expression of the extra-functional requirements of each step of the use case descriptions. Since the requirements can be very diverse (e.g. maximal allowed execution time, memory consumption, etc.), the *efp* annotation contains its type, value, and unit. As arise from its name, the *efp* annotation enables to test the extra-functional properties. The form of the annotations is depicted in Fig. 2.

```
<method: name; param1,...,
param2; result>
<create: name>
<use: name>
<efp: type; value; unit>
```

FIG. 2 THE ANNOTATIONS FORM

After the addition of our annotations, the use case descriptions are transformed into the OBA, which now incorporates the new annotations. The OBA can be regarded as an oriented graph. For the generation of the testing scenarios, it is necessary to explore all paths from the beginning of each use case to its end. A modified depth-first search (DFS) algorithm (Even 2000) can be used for this purpose. These paths represent all different flows of the program. Thanks to the *method* and *create-use* annotations, the flow is directly linked to the methods and artefacts. So, it is possible to generate the sequence of method invocations. The *efp* annotations can be then used for the testing of their extra-functional properties.

Interface-based Scenarios Generation

The second approach is the generation of the skeletons

of the testing scenarios from the known interfaces of particular tested components. In this extreme case, we assume that the UML diagrams, behavioral description, and the source code are not at our disposal. The only known information about the tested components is their interfaces and dependencies (i.e. provided and required services). It should be noted that the approach described in following paragraphs does not utilize a reverse engineering as describe in (Naseer, Rehman, and Hussain 2010), since the interfaces of the tested components are public and can be obtained using standard services of the component framework.

Assume now that there is a complete component-based application constructed from several components with all their dependencies satisfied (i.e. there is no missing service required by the components). This entire application can be imported to the SimCo simulation tool. During this import, the intermediate components are introduced among the particular software components, since all of them are real.

Because the services of the particular software components are known, it is now possible to sequentially invoke these services one by one. If the service is invoked on a component without dependencies (i.e. the component does not require services of other components), the only consequences of the service invocation can be a change of the inner state of the component and a return value if there is any. However, if a service is invoked on a component, which requires services of other components, this invocation can (but does not have to) result into invocation of one or more services on other components. These subsequent service invocations can be easily intercepted using the intermediate components. So, for each invocation, it is possible to construct a tree of subsequent service invocations.

It should be noted that it is possible for a service invocation to cause different subsequent service invocations based on its parameters or even the (unknown) inner state of the software component, on which it is performed. An example can be a software component providing service for storing of a picture in selected format (*PictureStoring*). For this purpose, it utilizes a component providing the JPEG compression (*CompressionJPEG*) and a component providing the PNG compression (*CompressionPNG*). The dependencies of the components are depicted in Fig. 3. If the PNG compression format is selected (as a parameter of the invocation of the service for storing of a picture on the *PictureStoring* component), the *PictureStoring* component invokes the service of the *CompressionPNG* component. If the JPEG compression format is selected, the *PictureStoring* component invokes the service of the *CompressionJPEG* component (see Fig. 4).

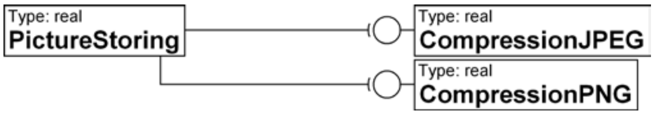

FIG. 3 THE COOPERATING SOFTWARE COMPONENTS

Hence, it is not sufficient to invoke each service only once. If the service has parameters, it is necessary to explore their possible values. For the enumeration types, this can be accomplished easily. However, it is more difficult for the basic types (integer, double) or even objects. For these types, it is usually impractical or even unfeasible to test all their possible values. In this case, the constraints of the values have to be set manually (e.g. using the documentation or another description of the software components if available).

If all the services of all the software components are tested the way described in previous paragraphs, we obtain a set of trees describing the consequences of the invocations of particular services. This set can be used as skeletons of the testing scenarios. Moreover, the external behaviors of all software components are then at least partially known. These behaviors can be used during preparation of the simulated components as replacement of some of the real component. Using the skeletons of the testing scenarios and the simulated components, the testing scenarios focused on the remaining real components can be (manually) finished.

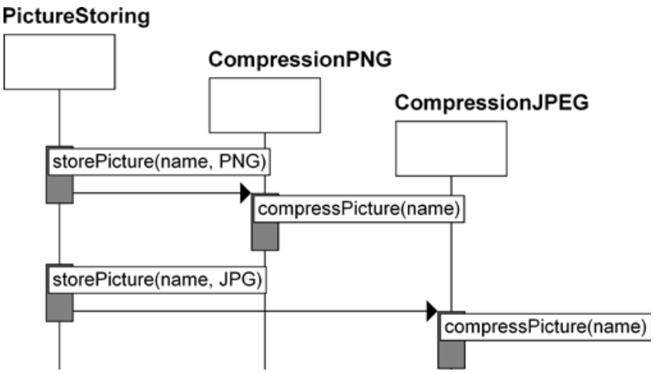

FIG. 4 DIFFERENT SERVICES INVOCATION BY ONE SERVICE

Case Study: Crossroad Control

In order to test the proposed approaches for the generation of the testing scenarios, the *Traffic crossroad control* case study is used. It represents a component-based application for the control of road traffic in a crossroad using traffic lights and variable traffic signs. It is expected to run on a specific hardware and

operate variety of sensors and control units (Potuzak, Lipka, Brada, and Herout 2012). However, this specific hardware is not important for the needs of this paper.

Case Study Description

The Traffic crossroad control application consists of nine components (see Fig. 5). Since the application is tested on a standard desktop computer, the components for handling of the specific hardware sensors (the *OpticDetection* and the *InductionLoop* components) and the control units (the *TrafficLightsController* and *VariableSignController* components) were replaced by the simulated components. Similarly, the *TrafficCrossroad* component, which provides information about the structure of the traffic crossroad, is simulated only. The reason is that its simulated version incorporates a road traffic simulation replacing the real traffic (Potuzak, Lipka, Brada, and Herout 2012), which is necessary for the testing of the majority of the components of the Traffic crossroad control application.

The remaining components can be either real or replaced by their simulated counterparts based on the actual needs. The *ControlPanel* component provides a user interface of the entire application. The *Traffic-ControlAlgorithm* component contains an algorithm for control of the traffic lights and the variable signs. For this purpose, it can require information from the sensors mediated by the *SensorAccess* component. This component is also used by the *StatisticCollector* component, which collects the data and provides various traffic statistics (Potuzak, Lipka, Brada, and Herout 2012).

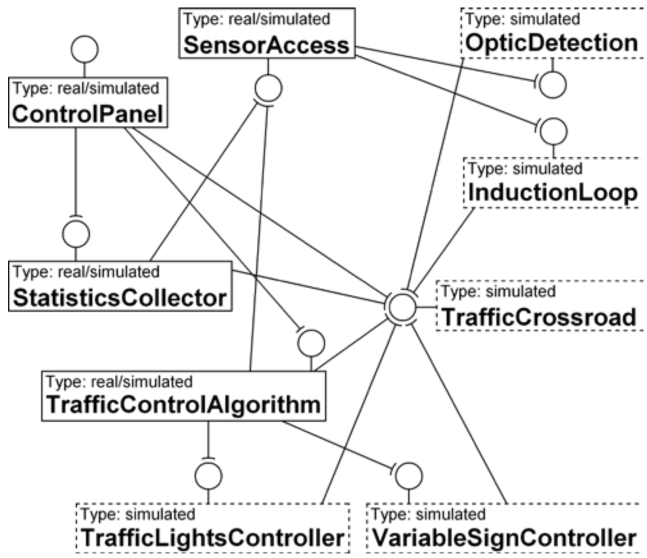

FIG. 5 DEPENDENCIES OF THE SOFTWARE COMPONENTS

Use-Case-based Scenarios Generation Example

Using the Traffic crossroad control case study, we are able to provide a small example of the generation of the testing scenarios based on the use case descriptions. Assume now that there is a use case description describing the change of the settings of the traffic control algorithm by the user (see Fig. 6).

```
Use case 1: Change traffic control
algorithm settings
1. The user opens dialog with traffic
   control algorithm settings
2. The system displays the traffic control
   algorithm current settings
   <method: ControlPanel.showSettings>
3. The user adjust the traffic control
   algorithm settings
4. The user confirms the traffic control
   algorithm settings
   <create: settings>
5. The system sets the new traffic control
   algorithm settings
   <use: settings>
   <method:
   TrafficControlAlgorithm.setSet-tings;
   #settings>
Variation 4a: New settings are invalid
4a1. The systems displays warning dialog
     <method: ControlPanel.showDialog>
4a2. The user closes the warning dialog
4a3. Goto step 3
```

FIG. 6 EXAMPLE OF A USE CASE DESCRIPTION

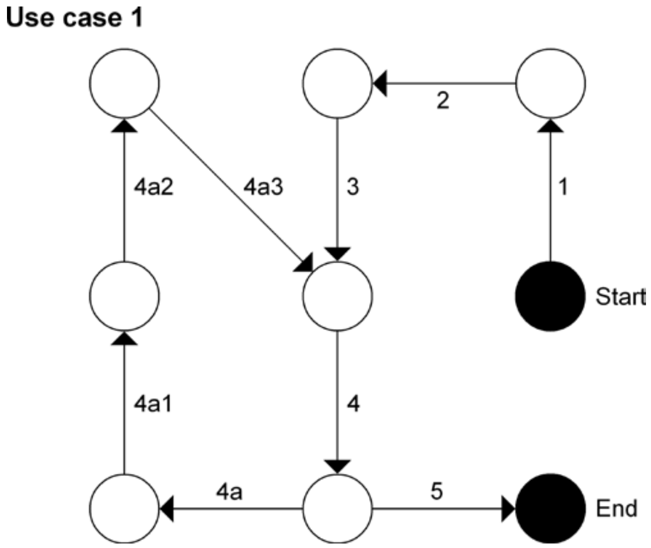

FIG. 7 EXAMPLE OF AN OVERALL BEHAVIORAL AUTOMATON

As can be seen in Fig. 6, the plain text of the use case description has been (manually) enriched with the annotations used for the generation of the testing scenarios. In order to keep the description legible, the annotations used by the Procasor tool directly are omitted. Using the Procasor tool, the use case description is transformed to an overall behavioral automaton, which is depicted in Fig. 7. This automaton incorporates the annotations necessary for the generation of the testing scenarios, although it does not used them. Using the modified DFS algorithm, two sequences of the service invocations can be generated (see Fig 8).

```
Sequence 1:
ControlPanel.showSettings();
TrafficControlAlgorithm.setSettings();
Sequence 2:
ControlPanel.showSettings();
ControlPanel.showDialog();
TrafficControlAlgoritm.setSettings();
```

FIG. 8 GENERATED SEQUENCES OF SERVICE INVOCATIONS

Interface-Based Scenarios Generation Example

Using the Traffic crossroad control case study, a small example can be provided for the generation of the skeletons of the testing scenarios as well. Assume now that all software components of the Traffic crossroad control application are imported to the SimCo simulation tool (see Fig. 5). The intermediate components are not depicted, but are present among the real components. All the components depicted with solid line are real and all the components depicted with dashed line are simulated. As it was stated before, all the components should be real for the interface-based scenarios generation to work. However, if it is known, that the simulated components do not have any dependencies (i.e. they do not require services of other software components), the generation can work properly as well.

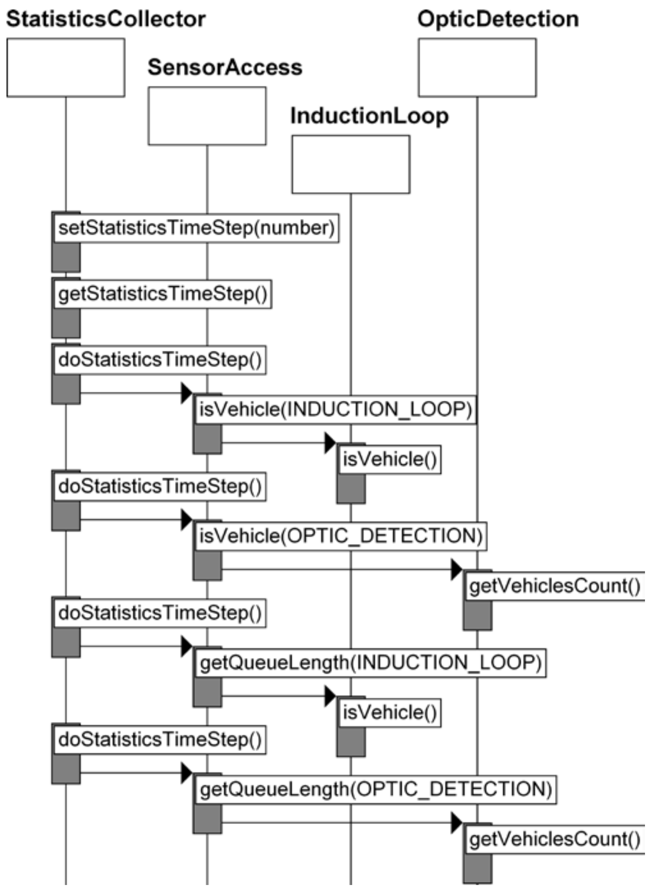

FIG. 9 PART OF A SEQUENCE OF THE SERVICES INVOCATIONS

Once the entire application is imported to the SimCo simulation tool, the particular services of the particular software components are invoked. The consequent invocations are traced using the intermediate components. A part of the sequence of the services invocations on the *StatisticsCollector* component is depicted in Fig. 9.

As can be seen in Fig. 9, the invocation of the same service with a different parameter can lead to subsequent invocation of different services. The service with the *number* parameter is problematic, because it is not feasible to test all possible values of this parameter. Fortunately, in this case, this particular service invocation seems not to cause any subsequent service invocation. However, this is not guaranteed, because all possible values were not tested – there is only one invocation of this service depicted in Fig. 9. In order to partially solve this issue, a list of values to set or the constraints of the values must be provided manually.

A complete set of sequences of the services invocations on all software components (a part is depicted in Fig. 9) is the basis for the skeletons of the testing scenarios.

Future work

The course of our future work is briefly described in following subsections.

Implementation and Testing of Described Methods

Since both described approaches are in a preliminary stage of development, as part of our future research, we will continue to work on the implementation of both approaches and their refinement. Once the implementations will be working properly, we will focus on the complex testing of both methods using the described and other case studies.

Comparison of Described and Other Methods

Both methods will be compared each other and also with other similar methods developed by other parties. The comparison will be focused on the amount of required manual work, the amount of required input data and its general availability, and the results of the particular methods.

Utilization of Reverse Engineering

We will also explore other possible approaches for the generation of the testing scenarios more thoroughly. For example, the utilization of the reverse engineering seems to be a viable way how to improve results and capabilities of the interface-based generation of the skeletons of the testing scenarios method.

Conclusion

In this paper, we discussed the possible approaches to semi-automated generation of the testing scenarios for simulation testing of software components, sets of software components, and entire component-based applications. We described the existing options as well as two newly developed approaches – the use-case-based generation of the testing scenarios and the interface-based generation of the skeletons of the testing scenarios.

The former approach is utilizable if the use case descriptions of the particular tested software components are at our disposal. The latter approach is utilizable even if there are only the interfaces and dependencies of the tested software components at our disposal. Both approaches are semi-automated in a sense that they are able to perform much of the work automatically, but manual input of an operator is still required. Both approaches are currently in a preliminary stage of development.

ACKNOWLEDGMENT

This work is supported by the Grant Agency of the Czech Republic under grant "Methods of development and verification of component-based applications using natural language specifications," Czech Science Foundation (GACR) 103/11/1489.

REFERENCES

Bauer, J., and Finger, A. "Test Plan Generation Using Formal Grammars." In Proceedings of the Fourth International conference on Software Engineering, Los Alamitos, 425–432, 1979.

Becker, S., Koziolek, H., and Reussner, R. "The Palladio component model for model-driven performance prediction." In Journal of Systems and Software, Vol. 82, No. 1, 3–22, 2009.

Bures, T., Hnetynka, P. and Plasil, F. "SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model." In Fourth International Conference on Software Engineering Research, Management and Applications, 2006.

Cockburn, A. Writing Effective Use Cases. Addison-Wesley, Boston, 2000.

De Santiago Jr., V. A., Vijaykumar, N. L. "Generating model-based test cases from natural language requirements for space application software." In Software Quality Journal, Vol. 20, No. 1, 77–143, 2012.

Even, S. Graph Algorithms (2nd edition). Cambridge University Press, 2000.

Fujimoto, R. M. Parallel and Distributed Simulation Systems. John Wiley & Sons, New York, 2000.

Heam, P. C., Kouchnarenko, O., and Voinot, J. "Component Simulation-based Substitutivity Managing QoS Aspects." In Electronic Notes in Theoretical Computer Science, Vol. 260, 109–123, 2010.

Hou, X., Wang, Y., Zheng, H., and Tang, G. "Integration Testing System Scenarios Generation Based on UML." In 2010 International Conference on Computer, Mechatronics, Control and Electronic Engineering, 271–273, 2010.

Lipka, R., Potuzak, T., Brada, P., and Herout, P. "Verification of SimCo – Simulation Tool for Testing of Component-based Application." In EUROCON 2013, Zagreb, 467–474, 2013.

Liu, S., Shen, W. "A Formal Approach to Testing Programs in Practice." In 2012 International Conference on Systems and Informatics, Yantai, 2509–2515, 2012.

Naseer, F., Rehman, S. U., Hussain, K. "Using Meta-data Technique for Component Based Black Box Testing." In 2010 6th International Conference on Emerging Technologies, Islamabad, 276–281, 2010.

The OSGi Alliance. "OSGi Service Platform Core Specification." release 4, version 4.2, 2009.

Potuzak, T., Lipka, R., Brada, P., and Herout, P. "Testing a Component-based Application for Road Traffic Crossroad Control using the SimCo Simulation Framework." In 38th Euromicro Conference on Software Engineering and Advanced Applications, Cesme, Izmir, 175–182, 2012.

Rubio, D. Pro Spring Dynamic Modules for OSGi™ Service Platform. Apress, USA, 2009.

Sapna, P. G., and Mohanty, H. "Automated Scenario Generation based on UML Activity Diagrams." In International Conference on Information Technology, 2008, 209–214, 2008.

Shirole, M., Kumar, R. "UML Behavioral Model Based Test Case Generation: A Survey." In ACM SIGSOFT Software Engineering Notes, Vol. 28, No. 4, 1–12, 2013.

Simko, V., Hauzar, D., Bures, T., Hnetynka, P., and Plasil, F. "Verifying Temporal Properties of Use-Cases in Natural Language." In Springer-Verlag, Vol. 7741, 350– 367, 2013.

Simko, V., Hnetynka, P., Bures, T., and Plasil, F. "FOAM: A Lightweight Method for Verification of Use-Cases." In 38th Euromicro Conference on Software Engineering and Advanced Applications, 228–232, 2012.

Simko, V., Hnetynka, P., Bures, T., and Plasil, F. Formal Verification of Annoted Use-Cases. Technical report 2012/2, Charles University in Prague, 2012.

Somé, S. S., Cheng, X. An Approach for Supporting System-level Test Scenarios Generation from Textual Use Cases. In Proceedings of the 2008 ACM symposium on Applied computing, Fortaleza, 724–729, 2008.

Szyperski, C., Gruntz, D., and Murer. S. Component Software – Beyond Object-Oriented Programming. ACM Press, New York, 2000.

Yongfeng, Y., Bin, L., Minyan, L., Zhen, L. "Test Cases Generation for Embedded Real-time Software Based on Extended UML." In 2009 International Conference on Information Technology and Computer Science, Kiev, 69–74, 2009.